Address-Indexed Memory Disambiguation and Store-to-Load Forwarding

Sam S. Stone, Kevin M. Woley, Matthew I. Frank
Department of Electrical and Computer Engineering
University of Illinois, Urbana-Champaign
{ssstone2,woley,mif}@uiuc.edu

Abstract

This paper describes a scalable, low-complexity alternative to the conventional load/store queue (LSQ) for superscalar processors that execute load and store instructions speculatively and out-of-order prior to resolving their dependences. Whereas the LSQ requires associative and age-prioritized searches for each access, we propose that an address-indexed store-forwarding cache (SFC) perform store-to-load forwarding and that an address-indexed memory disambiguation table (MDT) perform memory disambiguation. Neither structure includes a CAM. The SFC behaves as a small cache, accessed speculatively and out-of-order by both loads and stores. Because the SFC does not rename in-flight stores to the same address, violations of memory anti and output dependences can cause in-flight loads to obtain incorrect values from the SFC. Therefore, the MDT uses sequence numbers to detect and recover from true, anti, and output memory dependence violations.

We observe empirically that loads and stores that violate anti and output memory dependences are rarely on a program's critical path and that the additional cost of enforcing predicted anti and output dependences among these loads and stores is minimal. In conjunction with a scheduler that enforces predicted anti and output dependences, the MDT and SFC yield performance equivalent to that of a large LSQ that has similar or greater circuit complexity. The SFC and MDT are scalable structures that yield high performance and lower dynamic power consumption than the LSQ, and they are well-suited for checkpointed processors with large instruction windows.

1. Introduction

Modern superscalars expose parallelism by executing instructions speculatively and out of order, without violating the data dependences among the instructions. To expose additional instruction-level parallelism, processors may execute load and store instructions prior to resolving their dependences. Such processors include mechanisms to detect and recover from memory dependence violations, to forward values from in-flight stores to dependent loads, and to buffer stores for in-order retirement. As instruction windows expand and the number of in-flight loads and stores increases, the latency and dynamic power consumption of store-to-load forwarding and memory disambiguation become increasingly critical factors in the processor's performance.

Conventional processors use a load/store queue (LSQ) to perform store-to-load forwarding, memory disambiguation, and in-order store retirement. The LSQ is neither efficient nor scalable. Store-to-load forwarding and memory disambiguation require fully associative, age-prioritized searches of the store queue and the load queue, respectively. In a conventional processor, these searches exhibit high latency and high dynamic power consumption. As the capacity of the LSQ increases to accommodate larger instruction windows, the latency and dynamic power consumption of associative LSQ searches threaten to become a performance bottleneck [22].

In our design the functions of store-to-load forwarding, memory disambiguation, and in-order retirement of stores are divided among three structures: an address-indexed store forwarding cache (SFC), an address-indexed memory disambiguation table (MDT), and a store FIFO. The SFC and MDT yield low latency and low dynamic power consumption for store-to-load forwarding and memory disambiguation. Because these structures do not include either content-addressable memories (CAM's) or priority encoders, they scale readily as the number of in-flight loads and stores increases.

The SFC is a small direct-mapped or set-associative cache to which stores write their values as they complete, and from which loads may obtain their values as they execute. Accessed in parallel with the L1 data cache, the SFC efficiently forwards values from in-flight stores to dependent loads. Both loads and stores access the SFC speculatively and out-of-order, but the SFC does not rename stores to the same address. Therefore, violations of true, anti, or output memory dependences can cause loads to obtain incorrect values from the SFC. The MDT detects all mem-

ory dependence violations via a technique similar to basic timestamp ordering [3], a traditional concurrency control technique in transaction processing. When a memory dependence violation is detected, the memory unit initiates recovery.

In this paper, we study the feasibility of replacing the LSQ with a store forwarding cache and a memory disambiguation table. Because the SFC does not perform memory renaming, we examine the effects of anti and output memory dependence violations and propose policies for managing those dependences. We make the following contributions:

1. Anti and output dependence violations increase in frequency as the number of in-flight loads and stores increases. However, loads and stores that violate anti and output dependences are rarely on a process's critical path. Enforcing predicted anti and output dependences among loads and stores incurs little additional complexity in the memory dependence predictor and the scheduler. However, enforcing these dependences greatly increases the performance of the SFC and the MDT.

2. In a conventional superscalar, the SFC and the MDT reduce the dynamic power consumption of store-to-load forwarding and memory disambiguation, while providing performance within 1% of an ideal LSQ. In a superscalar with a large instruction window, the SFC and MDT provide performance equivalent to that of a large LSQ that has similar or greater circuit complexity.

3. Because the CAM-free MDT and SFC scale readily, they are ideally suited for checkpointed processors with large instruction windows.

The remainder of the paper is organized as follows. In Section 2, we present the microarchitecture of the SFC and the MDT, along with a memory dependence predictor that enforces predicted true, anti, and output dependences. In Section 3 we present our simulation methodology and the results of our experiments. In Section 4, we discuss the motivation for low-power, low-latency memory disambiguation and store-to-load forwarding, as well as recent proposals for optimizing the memory subsystem. Section 5 concludes.

2. Design

We propose that the functions of store-to-load forwarding, memory disambiguation, and in-order retirement of stores be divided among three structures: an address-indexed store forwarding cache (SFC), an address-indexed memory disambiguation table (MDT), and a store FIFO. The address-indexed SFC and MDT yield low latency and low dynamic power consumption for store-to-load forwarding and memory disambiguation. Furthermore, these structures scale readily as the number of in-flight loads and stores increases.

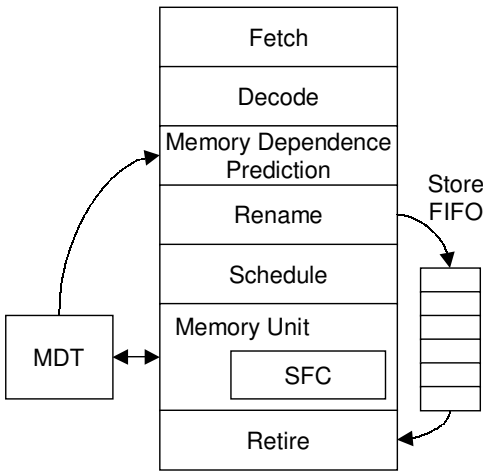

Figure 1. Processor Pipeline. A store enters the non-associative store FIFO at dispatch, writes its data and address to the FIFO during execution, and exits the FIFO at retirement. The Store Forwarding Cache (SFC) is part of the memory unit, and is accessed by both loads and stores. The Memory Disambiguation Table (MDT) interacts with the memory unit and memory dependence predictor.

The SFC is a small cache to which a store writes its value as it completes, and from which a load may obtain its value as it executes. Accessed in parallel with the L1 cache, the SFC efficiently bypasses values from in-flight stores to dependent loads. Because loads and stores access the SFC out of order, the accesses to a given address may violate true, anti, or output dependences. To detect memory dependence violations the MDT tracks the highest sequence numbers yet seen of the loads and stores to each in-flight address. When the ordering of sequence numbers indicates that a memory dependence violation has occurred, the MDT initiates recovery.

In the foregoing discussion, we assume that the processor includes a simple instruction re-execution mechanism, whereby the memory unit can drop an executing load or store and place the instruction back on the scheduler's ready list. In particular, loads and stores that are unable to complete because of structural conflicts in the SFC or the MDT may be re-executed. Because the scheduler does not speculatively issue instructions dependent on a load before the load completes, this mechanism is considerably simpler than selective re-execution.

In Section 2.1, we describe a memory dependence predictor and scheduler that enforce predicted true, anti, and output memory dependences. Sections 2.2 and 2.3 present

the memory disambiguation table and the store forwarding cache, respectively. Section 2.4 discusses optimizations related to recovering from memory ordering violations and structural conflicts in the MDT and SFC.

2.1. Scheduling Predicted Dependences

Because the SFC does not perform memory renaming, anti and output memory dependence violations can potentially degrade the processor's performance. To avoid unnecessary pipeline flushes, we adapt the store-set predictor [5] and the scheduler to enforce predicted true, anti, and output memory dependences. The *producer-set* predictor includes a producer table and a consumer table (in place of the store-set id table), as well as a last-fetched producer table (in place of the last-fetched store table). When the MDT notifies the producer-set predictor of a dependence violation, the predictor inserts a dependence between the earlier instruction (the producer) and the later instruction (the consumer) by placing the two instructions in the same producer set. That is, the predictor places the same producer-set id in the producer and consumer table entries indexed by the producer and consumer PC's, respectively. Rules for merging producer sets are identical to the rules for merging store sets.

When a load or store instruction enters the memory dependence predictor, it accesses the PC-indexed producer and consumer tables (PT and CT, respectively). If the instruction finds a valid producer-set id in the producer table, it uses that id to access the corresponding entry in the last-fetched producer table (LFPT), obtains a dependence tag from the LFPT's free list, and writes the dependence tag number into the LFPT entry. Likewise, if the instruction finds a valid producer-set id in the consumer table, it uses that id to read the dependence tag number from the corresponding LFPT entry.

The scheduler tracks the availability of dependence tags in much the same manner as it tracks the availability of physical registers. A load or store instruction is not permitted to issue until the dependence tag that it consumes (if any) is ready. After the scheduler issues a load or store instruction to the memory unit, the scheduler marks the dependence tag produced by the instruction (if any) as ready. In this manner, predicted consumers of a producer set become dependent on that set's most recently fetched producer.

In conjunction with an MDT, the producer-set predictor enforces predicted true, anti, and output memory dependences. In conjunction with an LSQ, the producer-set predictor behaves much like the conventional store-set predictor, with one exception. The producer-set predictor does not enforce predicted output dependences among stores in the same producer set, because the LSQ does not alert the dependence predictor when output dependence violations occur. By enforcing predicted output dependences, the store-set predictor minimizes the number of false memory ordering violations caused by silent stores [18]. When a store completes after a subsequent store and load to the same address have completed, a conventional LSQ detects a memory ordering violation, even though the load obtained the correct value from the later store. We simulate an LSQ that does not falsely flag silent stores as ordering violations; therefore, the producer-set dependence predictor is well-suited to both the LSQ and the MDT.

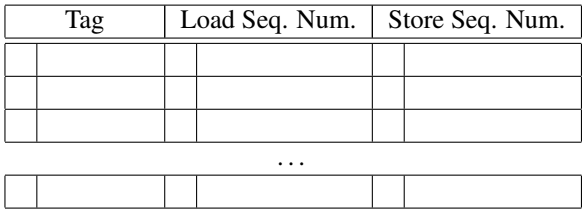

Figure 2. The Memory Disambiguation Table is a set associative cache-like structure. It stores the highest sequence numbers yet seen of in-flight loads and stores to the address indicated by the tag. As a load or store instruction completes, the MDT compares the instruction's sequence number to the sequence numbers in the corresponding MDT entry. If the sequence number of the completing instruction is lower than the sequence numbers in the MDT, there may be a dependence violation.

2.2. Memory Disambiguation Table (MDT)

The memory disambiguation table (Figure 2) is an address-indexed, cache-like structure that replaces the conventional load queue and its associative search logic. To reduce the complexity of memory disambiguation, the MDT buffers the sequence numbers of the latest load and store to each in-flight memory address. Therefore, memory disambiguation requires at most two sequence number comparisons for each issued load or store. When the MDT detects a memory dependence violation, it conservatively initiates recovery by flushing all instructions subsequent to the load or store whose late execution caused the dependence violation.

The MDT uses sequence numbers to detect memory dependence violations. Conceptually, the processor assigns sequence numbers that impose a total ordering on all in-flight loads and stores. Techniques for efficiently assigning sequence numbers to loads and stores (and for handling sequence number overflow) are well known. For example

the LSQ relies on sequence numbers to locate the slot from which a load (or store) should initiate its search of the store (or load) queue [1, 6, 11].

When a load issues, it calculates its address and uses the low-order address bits to access its entry in the MDT. If the load's sequence number is later than the load sequence number in the MDT entry, or if the entry contains no valid load sequence number, then the load must be the latest issued load to its address. Therefore, the load writes its sequence number into the MDT. Otherwise, the load sequence number in the MDT entry does not change. If the load's sequence number is earlier than the store sequence number in the MDT, the MDT detects an anti-dependence violation. Otherwise, no memory ordering violation is known to have occurred. Finally, when the load retires, it compares its sequence number to the load sequence number in its MDT entry. If the sequence numbers match, then the retiring load is the latest in-flight load to its address. Thus, the MDT invalidates the entry's load sequence number by clearing the associated valid bit. If the entry's store sequence number is also invalid, then the MDT frees the entry.

A store accesses the MDT in a similar manner. If the store's sequence number is later than the store sequence number in the MDT, or if the entry contains no valid store sequence number, then the store writes its sequence number into the MDT. Otherwise, the MDT detects an output dependence violation. Likewise, if the store's sequence number is earlier than the load sequence number, the MDT detects a true dependence violation. Retirement of stores is analogous to retirement of loads.

When the MDT detects a memory dependence violation, it conservatively flushes all instructions subsequent to the load or store whose late execution caused the violation. In the case of a true dependence violation, the pipeline flushes all instructions subsequent to the store that executed later than a dependent load, and the dependence predictor inserts a dependence between the store (producer) and the load (consumer). To facilitate communication between the MDT and the dependence predictor, each MDT entry holds not only the sequence numbers of the latest load and store to each in-flight address, but also pointers to those same instructions (e.g., pointers to the instructions' reorder buffer slots).

The case for an output dependence violation is analogous, with the pipeline flushing all instructions subsequent to the earlier store, and the dependence predictor inserting a dependence between the earlier store (producer) and the later store (consumer). Finally, when an anti dependence violation occurs, the pipeline flushes the load and all subsequent instructions, and the dependence predictor inserts a dependence between the earlier load (producer) and the later store (consumer).

Entries in the MDT may be tagged or untagged. In an untagged MDT, all in-flight loads and stores whose addresses map to the same MDT entry simply share that entry. Thus, aliasing among loads and stores with different addresses causes the MDT to detect spurious memory ordering violations. Tagged entries prevent aliasing and enable construction of a set-associative MDT. With a tagged MDT, if a set conflict prevents an in-flight load or store from allocating an entry in the MDT, the memory unit drops the instruction and places it back on the scheduler's ready list.

When a memory instruction is unable to complete because it cannot allocate an entry in the tagged MDT (or in the SFC, which is also tagged), reorder buffer lockup becomes a concern. If later loads or stores have allocated the MDT/SFC entries needed by this instruction, then those entries will not be available even when this instruction reaches the head of the ROB. To avoid ROB lockup, the memory unit permits any load or store at the head of the ROB to execute without accessing the MDT or the SFC. If the instruction is a load, it simply obtains its value from the cache-memory hierarchy and retires. If the instruction is a store, it writes its value to the store FIFO and retires.

The number of bytes tracked by each MDT entry (the MDT granularity) is an important parameter. Increasing the MDT's granularity allows each MDT entry to disambiguate references to a greater number of memory bytes. In a small, tagged MDT, increasing the MDT's granularity can reduce the number of tag conflicts, thereby reducing the number of loads and stores that must be re-executed. However, increasing the granularity of the MDT also increases the probability of detecting spurious memory ordering violations among accesses to the distinct addresses that map to a single MDT entry. Because the penalty for a memory ordering violation (a pipeline flush) is more severe than the penalty for a structural conflict (a re-execution), it is desirable to build a large MDT with small granularity. Empirically, we observe that an 8-byte granular MDT is adequate for a 64-bit processor.

Finally, when a partial pipeline flush occurs, the MDT state does not change in any way. Partial flushing occurs frequently in architectures that checkpoint the register alias table (RAT), such as the Alpha-21264 [15] and the MIPS-R10000 [26]. When a partial flush occurs, the processor recovers to the nearest checkpoint. Instructions later than the nearest checkpoint are flushed, and fetch resumes along the correct path. The LSQ recovers from partial pipeline flushes simply by adjusting its tail pointers to discard the canceled instructions. However, previous address-indexed LSQ's have suffered from severe penalties for partial flushing [22]. By ignoring partial flushes, the MDT simply becomes conservative. That is, the MDT may detect spurious memory ordering violations with respect to canceled loads and stores. However, our experiments show that the associated penalty is not severe.

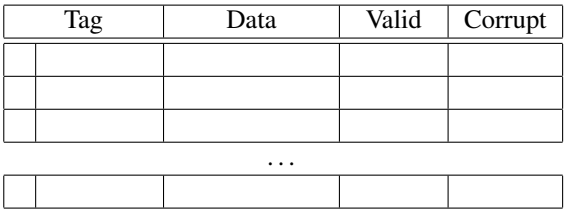

Figure 3. The Store Forwarding Cache is a small, tagged, set-associative cache that holds speculative store values that have not yet committed to the memory system. Each line holds up to 8 bytes of data. The valid bits are used to detect partial matches from sub-word stores. The corruption bits are used during partial pipeline flushes to avoid forwarding data from canceled stores.

2.3. Store Forwarding Cache (SFC)

The store forwarding cache (Figure 3) is an address-indexed, cache-like structure that replaces the conventional store queue's associative search logic. In the absence of a CAM, the store queue becomes a simple FIFO that holds stores for in-order, non-speculative retirement. The SFC reduces the dynamic power consumption and latency of store-to-load forwarding by buffering a single, cumulative value for each in-flight memory address, rather than successive values produced by multiple stores to the same address. A load obtains a value from the SFC by performing an address-indexed lookup, rather than by searching a queue for earlier matching stores.

The SFC is a small, tagged cache. Each SFC entry includes eight bytes of data, an eight-bit valid mask, and an eight-bit corruption mask. The data bytes hold the in-flight value of an aligned memory word. When a store executes, it calculates its address and performs a tag check in the SFC. If the store's address is already in the SFC, or if an entry in the address's set is available, the store writes its data to that entry, sets the bits of the valid mask that correspond to the bytes written, and clears the same bits of the corruption mask. The purposes of the valid and corruption masks are discussed below. Finally, the SFC frees an entry whenever the latest store to the entry's address retires. This condition is identical to the MDT's condition for invalidating an entry's store sequence number. When the SFC frees an entry, it resets the entry's valid and corruption masks, as well as the tag valid bit.

When a load executes, it accesses the SFC in parallel with the L1 cache. If the load finds a matching address in the SFC, and all the bytes accessed by the load are valid (a full match), then the load obtains its value from the SFC. By contrast, if only a subset of the bytes needed by the load are marked valid (a partial match), the memory unit either places the load back in the scheduler or obtains the missing bytes from the cache. Finally, if any of the bytes needed by the load are corrupt, the memory unit places the load back in the scheduler.

The corruption bits guard SFC data that has been or may have been corrupted by canceled store instructions. While the MDT does detect all memory dependence violations, it does not account for the effects of executed store instructions that are later canceled due to branch mispredictions or dependence violations. If canceled and non-canceled loads and stores to the same address issue in order, and the SFC forwards a value from a canceled store to a non-canceled load, then the MDT will not detect a violation.

When a full pipeline flush occurs the memory unit simply flushes the SFC, thereby discarding the effects of canceled stores. By contrast, when a partial pipeline flush occurs the memory unit cannot flush the SFC, because the pipeline still contains completed stores that were not flushed and have not been retired. Such stores have not committed their values to the cache, and canceled stores may have overwritten their values in the SFC. The addresses of such stores are said to be corrupt because their values are not known to be anywhere in the SFC-cache-memory hierarchy.

To prevent loads from obtaining stale values from the cache or erroneous values from canceled stores, each SFC entry includes a corruption mask. During a partial flush, the SFC overwrites each entry's corruption mask with the bitwise OR of its valid mask and its existing corruption mask. That is, the SFC marks every byte that is valid as corrupt. If a load attempts to access a corrupt address, then the memory unit drops the load and places the instruction back in the scheduler's ready list.

For example, assume that the following sequence of instructions (including an incorrectly predicted branch) has been fetched and dispatched to the scheduler.

```
[1] ST M[B000] <- A1A1
[2] LD R1      <- M[B000]
    BRANCH (mispredicted)
[3] ST M[B000] <- B2B2
```

Assume store [3] (along the incorrect path) executes before the branch instruction. Then store [3] will write the value `B2B2` into the SFC over the value `A1A1` that was written by store [1]. When the branch instruction is finally executed (and found to be mispredicted) the machine will perform a partial flush of those instructions following the branch, including store [3]. When the partial flush occurs the SFC marks every byte that is valid as corrupt, including the SFC entry corresponding to the bytes of address `B000`.

If a load instruction

```
[4] LD R2      <- M[B000]
```

is now fetched along the correct path and attempts to read its value from the SFC it will find the entry marked as corrupt. The memory unit does not complete the load, but rather signals the scheduler to place the load instruction back in the ready queue.

Eventually store [1] will retire and commit its value to the cache. When store [3] retires both the corrupt and valid bits in the SFC entry for address `B000` will be cleared. Load [4] will then execute and obtain the value of store [1] from the cache.

2.4. Recovery

The recovery policy in LSQ-based systems is limited to a single case. The LSQ does not suffer from anti or output dependence violations, because it renames in-flight stores to the same address. When the LSQ detects a true dependence violation, the load queue supplies the PC of the earliest load that violated a true dependence with respect to the executing store instruction. Thus, the associative load queue search not only disambiguates memory references, but also permits aggressive recovery from true dependence violations. Assuming that the system created a checkpoint when the misspeculated load dispatched, the load queue enables the processor to recover from a true dependence violation by flushing the earliest conflicting load and all subsequent instructions.

By contrast, in a memory subsystem based on the MDT/SFC, true, anti, and output dependence violations all necessitate some form of recovery. As described in Section 2.2, the memory subsystem's policies for recovering from memory ordering violations are conservative. We propose more aggressive policies for recovering from ordering violations and structural conflicts. These policies offer tradeoffs between performance and complexity.

2.4.1 Recovery from True Dependence Violations

When the MDT detects a true dependence violation, only the latest matching load is buffered in the MDT. If multiple in-flight loads that violate the true dependence have issued, then the MDT certainly does not contain a pointer to the earliest conflicting load. Thus, the conservative recovery policy flushes all instructions subsequent to the completing store.

However, if the load that violates the true dependence is the only in-flight load to its address, then that load must be the latest conflicting load. This observation permits a less conservative recovery policy. That is, each MDT entry could keep a count of the number of loads completed but not yet retired. When the MDT detects a true dependence violation, if this counter's value is one, the processor can flush the early load and subsequent instructions, rather than the instructions subsequent to the completing store.

2.4.2 Recovery from Output Dependence Violations

When the MDT detects an output dependence violation, the executing store has overwritten the value of a later store to the same address in the SFC. If a later load to the same address attempted to access that entry, the load would obtain the wrong value, but the MDT would not detect a second dependence violation. Thus, when the MDT detects an output dependence violation, the conservative recovery policy flushes all instructions subsequent to the completing store.

This policy is conservative because the pipeline flush occurs before any load obtains the corrupted value from the overwritten SFC entry. Upon detection of an output dependence violation, the memory subsystem could simply mark the corresponding SFC entry as corrupt, and optionally alert the memory dependence predictor to insert a dependence between the offending stores. The normal policies for handling corruptions (as described in Section 2.3) would then apply.

2.4.3 Recovery from Structural Conflicts

When a load or store is unable to complete because of a set conflict or an SFC corruption, the memory unit squashes the instruction and places it back in the scheduler. However, as the instruction's source registers are ready, the scheduler may re-issue the instruction after just a few clock cycles, even though the set conflict or SFC corruption likely still exists. To prevent the same instruction from continually re-executing, the scheduler could add a single *stall bit* to each load and store instruction. When the memory unit places an instruction back in the scheduler, the scheduler sets that instruction's stall bit, indicating that it is not ready for issue. As a simple heuristic, the scheduler clears all stall bits whenever the MDT or SFC evicts an entry.

3. Evaluation

We used an execution-driven simulator of a 64-bit MIPS pipeline to evaluate the performance of the MDT and the SFC relative to the performance of the LSQ. The simulator executes all instructions, including those on mispredicted paths, and the results produced by retiring instructions are validated against a trace generated by an architectural simulator. We simulate two different processors, a baseline superscalar that supports up to 128 instructions in flight simultaneously and an aggressive superscalar that supports up to 1024 instructions in flight simultaneously. Both processors include Alpha-21264 style renaming and checkpoint recovery. Figure 4 lists the simulation parameters for both processors.

In both the baseline and the aggressive configurations, we model a highly idealized LSQ with infinite ports, infinite search bandwidth, and single-cycle bypass latency.

As discussed in Section 2.1 the LSQ does not falsely flag memory ordering violations caused by silent stores, and the producer-set predictor does not enforce output dependences among stores in a producer set.

In all experiments, the granularity of the MDT is eight bytes, and the data field of the SFC is eight bytes wide. Of the optimizations discussed in Section 2.4, we model only the optimized scheduling of re-executed loads and stores. To model the tag check in the SFC, we increase the latency of store instructions by one cycle for all experiments with the SFC. Likewise, to model the tag check in the MDT, we increase the penalty for memory ordering violations by one cycle with the MDT. Finally, the scheduler is idealized in the following way. When predicted producer loads and stores issue, the scheduler aggressively awakens their predicted consumers. However, the scheduler oracularly avoids awakening predicted consumers of loads and stores that will be replayed.

We simulated 19 of the 26 SPEC 2000 benchmarks, each with the lgred or mdred Minnesota Reduced inputs. Benchmarks were run to completion or to 300,000,000 instructions. We lack compiler support for the Fortran 90 benchmarks (`galgel`, `facerec`, `lucas` and `fma3d`) and runtime library support for `wupwise`, `sixtrack` and `eon`. For the aggressive processor, results for mesa were not available due to a performance bug in the simulator's handling of system calls.

3.1. Baseline Processor

We first evaluate the performance of the MDT/SFC in the baseline superscalar, using two different configurations of the memory dependence predictor. In the first configuration (labeled ENF in the figures), the dependence predictor inserts a dependence arc between a pair of instructions whenever the MDT detects any type of memory ordering violation. Therefore, the dependence predictor enforces predicted true, anti, and output dependences. In the second configuration (labeled NOT-ENF in the figures), the dependence predictor inserts a dependence arc between a pair of instructions only when the MDT detects a true dependence violation. Thus, the dependence predictor enforces only predicted true dependences.

As Figure 5 shows, when the memory dependence predictor enforces only true dependences, the MDT/SFC provides average performance within 3% of an idealized LSQ that has 48 entries in the load queue and 32 entries in the store queue. When the dependence predictor enforces true, anti, and output dependences, the MDT/SFC provides average performance within 1% of the 48x32 LSQ. Furthermore, the dependence predictor reduces the rate of anti and output dependence violations by more than an order of magnitude. The decreased rates of output dependence violations in gzip, vpr route, and mesa, yield significant increases in their respective IPC's.

As increasing the size of the LSQ does not increase the performance of any of the simulated benchmarks, we conclude that the MDT/SFC and the producer-set predictor represent a low-power, high-performance alternative to the conventional LSQ in modern superscalars. Likewise, we conclude that in the conventional superscalar, the producer-set predictor effectively enforces true, anti, and output dependences without unnecessarily constraining the out-of-order issue of loads and stores.

3.2. Aggressive Processor

Next, we evaluate the performance of the MDT/SFC in the aggressive superscalar. As before, we evaluate ENF and NOT-ENF configurations of the producer-set predictor for the MDT/SFC. In the ENF configuration, we alter the dependence predictor to enforce a total ordering upon loads and stores in the same producer set. The dependence predictor achieves this effect by treating any load or store involved in a dependence violation as both a producer and a consumer. We observe empirically that, in the aggressive superscalar, the policy of enforcing a total ordering on loads and stores in a producer set outperforms the policy of enforcing predicted producer-consumer relationships.

Relative to the NOT-ENF configuration, the average IPC of the ENF configuration is 14% higher across the specint benchmarks and 43% higher across the specfp benchmarks. Furthermore, across all benchmarks the average rate of memory dependence violations decreases from 0.93% in the NOT-ENF configuration to 0.11% in the ENF configuration. Clearly, the performance of the NOT-ENF configuration is severely restricted by its high rate of memory ordering violations.

As Figure 6 shows, relative to the 120x80 LSQ, the average IPC of the MDT/SFC is 9% lower across the specint benchmarks. An analysis of the benchmarks that perform poorly is given below. Nevertheless, given the highly idealized nature of the simulated LSQ, the relative performance of the MDT/SFC is quite encouraging. On the specfp benchmarks, the MDT/SFC yields performance 2% better than that of the 120x80 LSQ. We conclude that in an aggressive superscalar with a large instruction window, the producer-set predictor, SFC, and MDT provide performance equivalent to that of a large LSQ with similar or greater circuit complexity.

Among the specint benchmarks, bzip2, mcf, and vpr route suffer performance degradations of 15% or more relative to the 120x80 LSQ. Among the specfp benchmarks, ammp and equake suffer performance drops of 10% or more. These five benchmarks merit further scrutiny.

The poor performances of bzip2 and mcf are caused by

Parameter	Baseline	Aggressive
Pipeline Width	4 instr/cycle	8 instr/cycle
Fetch Bandwidth	Max 1 branch/cycle	Up to 8 branches/cycle
Branch Predictor	8Kbit Gshare	+ 80% mispredicts turned to correct predictions by an "oracle"
Memory Dep. Predictor	16K-entry PT and CT, 4K producer id's, 512-entry LFPT	
Misprediction Penalty	8 cycles	
MDT	4K sets, 2-way set assoc.	8K sets, 2-way set assoc.
SFC	128 sets, 2-way set assoc.	512 sets, 2-way set assoc.
Renamer	128 checkpoints	1024 checkpoints
Scheduling Window	128 entries	1024 entries
L1 I-Cache	8Kbytes, 2-way set assoc., 128 byte lines, 10 cycle miss	
L1 D-Cache	8Kbytes, 4-way set assoc., 64 byte lines, 10 cycle miss	
L2 Cache	512 Kbytes, 8-way set assoc., 128 byte lines, 100 cycle miss	
Reorder Buffer	128 entries	1024 entries
Function Units	4 identical fully pipelined units	8 units

Figure 4. Simulator parameters for the baseline and aggressive superscalar processors.

excessive SFC set conflicts and MDT set conflicts, respectively. In bzip2, over 50% of dynamic stores must be replayed because of set conflicts in the SFC. The rate of SFC set conflicts across all other specint benchmarks is less than 0.16%. Likewise, in mcf, over 16% of dynamic loads must be replayed because of set conflicts in the MDT. The rate of such set conflicts across all other specint benchmarks is only 0.002%.

Bzip2 and mcf are limited by the size, associativity, and hash functions of the SFC and MDT, respectively. At present, the hash functions use the least significant bits of the load/store address to select a set in the SFC or MDT. This simple hash makes the caches susceptible to high conflict rates when a process accesses multiple data structures whose size is a multiple of the SFC or MDT size. To confirm our intuition, we increased the associativity of the SFC and the MDT to 16 while maintaining the same number of sets. In this configuration, only 0.07% of bzip2's stores experience set conflicts in the SFC, and the IPC increases by 9.0%. Likewise, 0.00% of mcf's loads experience set conflicts in the MDT, and the IPC increases by 6.5%. We conclude that a better hash function or a larger, more associative SFC and MDT would increase the performance of bzip2 and mcf to an acceptable level.

By contrast, vpr route, ammp, and equake all experience relatively high rates of SFC corruptions. In these three benchmarks, roughly 20% of all dynamic loads must be replayed because of corruptions in the SFC. Most other benchmarks experience SFC corruption rates of 6% or less. Although the causes of the high corruption rates in these three benchmarks are unknown, it is likely that the corruption mechanism simply responds poorly to certain patterns of execution.

As an alternative to the corruption bits, the SFC could manage the effects of canceled stores by explicitly tracking their sequence numbers. For example, when a partial pipeline flush occurred, the SFC could record the sequence numbers of the earliest and latest instructions flushed (the *flush endpoints*). If the SFC attempted to forward a value from a canceled store, that store's sequence number would fall between the flush endpoints, and memory unit would place the load back in the scheduler's ready list. Of course, the performance of this mechanism would depend on the number of flush endpoints tracked by the SFC.

4. Background and Related Work

Independence prediction and dependence prediction can significantly increase the performance of an out-of-order memory subsystem. Memory independence predictors, such as the store-set predictor [5] and the MDPT/MDST [16], permit loads to aggressively issue prior to stores with unresolved addresses, while minimizing the rate of true memory dependence violations. By contrast, memory dependence predictors identify dependent store-load pairs so that data can be forwarded directly from the store (or the instruction that produced the store's value) to the load (or the instruction that consumes the load's value). Techniques for explicitly linking stores and dependent loads without incurring the overhead of an associative store queue search are described in [13, 25, 17].

Onder and Gupta propose an optimization of the conventional store-set predictor and LSQ [18]. They observe that the LSQ detects a false memory ordering violation if a silent store (a store that is overwritten by a later store before any

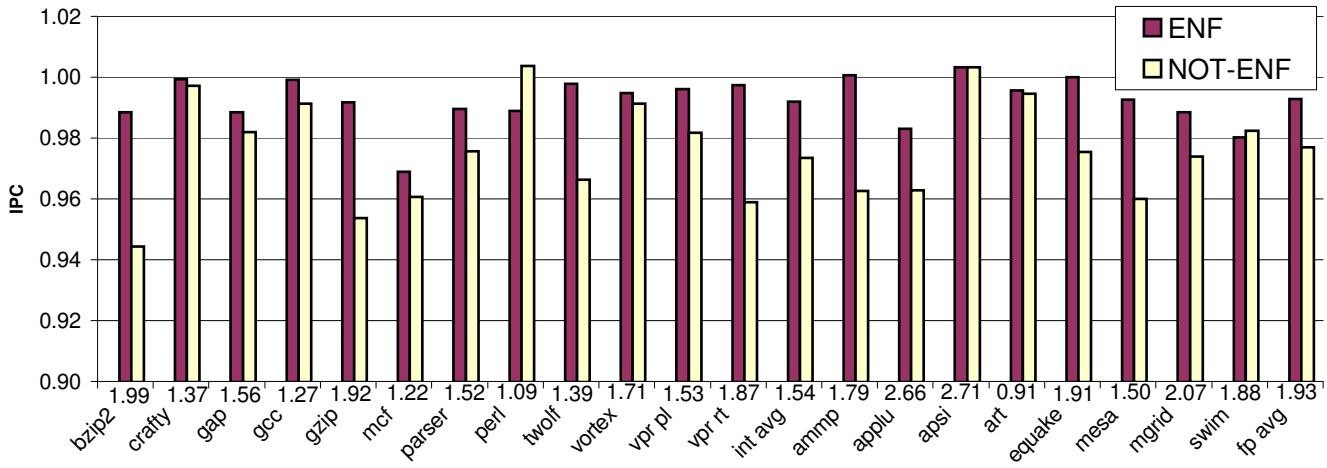

Figure 5. The Spec 2000 benchmarks running on a 4-wide baseline superscalar. The number at the bottom of the bars for each benchmark shows the IPC obtained using an aggressive LSQ with 48 entries in the load queue and 32 entries in the store queue. The LSQ does not suffer from false memory dependence violations. The Y-axis shows performance normalized to the IPC of the 48x32 LSQ. The left hand bar shows the normalized IPC obtained using a 256 entry, 2-way associative store forwarding cache, an 8192 entry, 2-way associative memory disambiguation table, and a producer-set predictor that enforces predicted true, anti, and output dependences. The right hand bar shows the IPC while running with the SFC and MDT, but with the producer-set predictor enforcing only predicted true dependences.

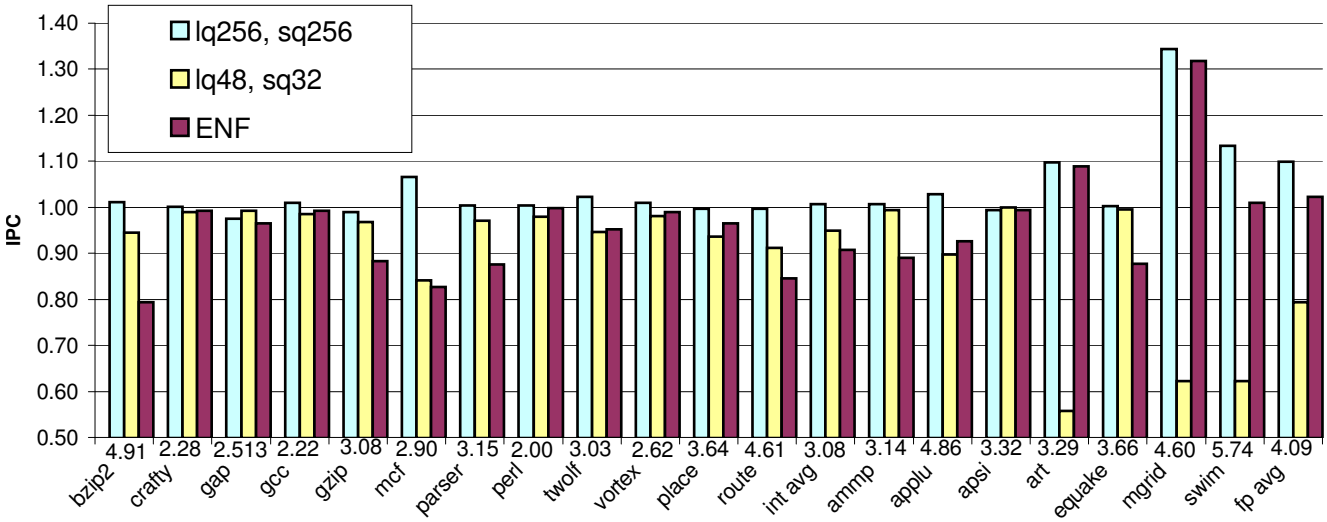

Figure 6. The Spec 2000 benchmarks running on an 8-wide aggressive superscalar. Results are normalized to the performance obtained using an idealized 120x80 LSQ. For each benchmark the left bar shows the normalized IPC obtained using an idealized 256x256 LSQ, while the middle bar shows the normalized IPC obtained using an idealized 48x32 LSQ. The right bar shows the normalized IPC obtained using a 1K entry, 2-way associative SFC, a 16K entry, 2-way associative MDT, and a producer-set predictor that enforces predicted true, anti, and output dependences.

load accesses the store's address) completes after a subsequent store and load to the same address have completed. When stores to the same address are permitted to issue out-of-order, these false violations can be frequent. Therefore, Chrysos and Emer's store-set predictor [5] constrains stores within the same store set to issue in order. At issue widths of 16 or more, this serialization of stores causes a severe drop in performance. To avoid this conservative enforcement of predicted output dependences, Onder and Gupta propose that memory disambiguation be delayed until retirement. As each load completes it stores the address and the value it read in a small fully associative buffer. When a store retires, it searches this associative buffer. If the store finds a load with matching address but a different value, it sets the load's exception bit. If the store finds a load with matching address and the same value, it clears the load's exception bit. A memory ordering violation occurs only if a load's exception bit is set when the load reaches the head of the ROB.

Cain and Lipasti [4] eliminate this associative load buffer search by replaying loads at retirement. At execution, a load accesses the data cache and the associative store queue in parallel. If the load issued before an earlier store with an unresolved address, then at retirement the load accesses the data cache again. If the value obtained at retirement does not match the value obtained at completion, then a memory dependence violation has occurred, and recovery is initiated.

Roth recently proposed the store vulnerability window [21], an optimization that reduces the number of loads that re-execute in systems that disambiguate memory references by re-executing loads at retirement, such as [4, 12], and in systems that perform redundant load elimination.

Although performing disambiguation at retirement does permit various optimizations of the load queue, the delay greatly increases the penalty for ordering violations. Furthermore, eliminating the associative store queue (as in our store forwarding cache) tends to increase the frequency of false memory ordering violations. We have observed empirically that the performance of a checkpointed processor with a large instruction window depends strongly on the frequency of memory ordering violations and the associated recovery penalty. In such processors, disambiguating memory references at completion is preferable.

Search filtering has been proposed as a technique for decreasing the LSQ's dynamic power consumption [19, 22]. Studies indicate that only 25% - 40% of all LSQ searches actually find a match in processors with large instruction windows [2, 22]. The goal of search filtering is to identify the 60% - 75% of loads and stores that will not find a match and to prevent those instructions from searching the LSQ. We view these techniques as orthogonal to the SFC and MDT. For example, search filtering could dramatically decrease the pressure on the MDT, thereby offering higher performance from a much smaller MDT.

To enable construction of high-capacity LSQ's with low search latencies, various segmented or hierarchical LSQ's have been devised. Both [19] and [1] propose segmenting and pipelining the LSQ. The segments can then be searched serially or concurrently. A parallel search achieves lower search latency at the expense of higher power consumption, while the latency and power consumption of a serial search depend on the number of segments searched.

The hierarchical store queue described in [2] includes a small level-1 store queue that holds the N most recently dispatched stores, a larger level-2 store queue that holds all other in-flight stores, and a membership test buffer that filters searches of the L2SQ. Both the L1 and L2 store queues include CAM's to support associative searches. By contrast, the hierarchical LSQ of [11] eliminates the CAM from the L2 store queue. In [11], the L2 LSQ uses a small cache for store-to-load forwarding, a simple FIFO for in-order buffering of stores, and an address-indexed, set-associative structure for memory disambiguation. These structures are similar to the SFC and MDT. However, the hierarchical LSQ maintains a conventional L1 LSQ, and uses the L2 LSQ only to track loads and stores in the shadow of a long-latency cache miss.

Torres proposes a distributed, hierarchical store queue in conjunction with a banked L1 data cache and a sliced memory pipeline [24]. The hierarchy comprises small L1 store queues that hold the N stores most recently dispatched to each L1 data cache bank, and a centralized L2 store queue that handles overflows from the L1 queues. A bank predictor steers loads and stores to the appropriate cache bank and store queue.

Timestamp based algorithms have long been used for concurrency control in transaction processing systems. The memory disambiguation table in our system is most similar to the *basic timestamp ordering* technique proposed by Bernstein and Goodman [3]. More sophisticated multiversion timestamp ordering techniques [20] also provide memory renaming, reducing the number of false dependences detected by the system at the cost of a more complex implementation.

Knight's Liquid system [14] was an early thread-level speculation system that allowed loads to speculatively bypass stores that they might depend on. Knight's system kept speculative stores in age order by assigning implicit sequence numbers to the caches of a shared-memory multiprocessor. The snooping hardware was modified so that writes only affected caches with higher sequence numbers and reads only read from caches with lower sequence numbers. Age-prioritized associative searches for either store-to-load forwarding or for memory ordering validation could then be performed by using the snooping hardware.

The influential Multiscalar architecture [8, 23] forcefully demonstrates the benefits of allowing loads to speculatively bypass stores. Multiscalar validates memory dependence speculations using an address resolution buffer (ARB) [9] that is similar to a hardware implementation of multiversion timestamp ordering. The ARB performs renaming, but is address ordered in the sense that multiple loads and stores to the same address all index into the same ARB row. Each ARB row is then ordered by sequence number and performs all three of the functions of the load-store queue: stores are retired in order, loads search the row in priority-encoded order to find forwarded store values, and stores search the row in priority-encoded order to validate load order. More recent efforts in the thread-level speculation domain have refocused on Knight's idea of modifying shared memory cache coherence schemes to support memory dependence speculation. An early example is [7].

The idea of speculatively allowing loads to bypass stores has also been examined in the context of statically scheduled machines. For example, Gallagher *et al* proposed a small hardware set-associative table, called a memory conflict buffer (MCB), that holds recently speculated load addresses and provides single cycle checks on each subsequent retiring store instruction [10]. A structure similar to the MCB is included in the IA-64.

5. Conclusion

As the capacity of the load/store queue increases to accommodate large instruction windows, the latency and dynamic power consumption of store-to-load forwarding and memory disambiguation threaten to become critical performance bottlenecks. By decomposing the LSQ into its constituent functions and dividing those functions among scalable structures, we can decrease the latency, complexity, and dynamic power-consumption of store-to-load forwarding and memory disambiguation.

In this paper, we have described a memory dependence predictor, a store forwarding cache, and a memory disambiguation table that yield performance comparable to that of an ideal LSQ. We have shown that in a conventional superscalar, the predictor, MDT, and SFC yield a 1% decrease in performance relative to the ideal LSQ, while eliminating the load queue and all of the LSQ's associative search logic. Furthermore, we have shown that in an aggressive superscalar with a large instruction window, the predictor, MDT, and SFC yield performance equivalent to that of a large LSQ with similar or greater circuit complexity. Finally, we have shown that loads and stores that violate anti and output dependences are rarely on a process's critical path. Enforcing such dependences incurs little additional complexity in the memory dependence predictor, but greatly increases the performance of the SFC and MDT.

Acknowledgments

The authors thank Thomas R. Novak for his help with a preliminary version of this work, and the members of the University of Illinois Implicitly Parallel Architectures group for their support in porting benchmarks and debugging the simulator used in this study. This research was supported in part by NSF grant CCF-0429711. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship.

References

[1] H. Akkary and K. Chow. Processor having multiple program counters and trace buffers outside an execution pipeline, 2001. U.S. Patent Number 6,182,210.

[2] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO 36*, page 423, 2003.

[3] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the Sixth International Conference on Very Large Data Bases*, pages 285–300, Montreal, Canada, Oct. 1980.

[4] H. W. Cain and M. H. Lipasti. Memory ordering: A value-based approach. In *ISCA-31*, pages 90–101, 2004.

[5] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *25th International Symposium on Computer Architecture (ISCA-25)*, pages 142–153, Barcelona, Spain, June 1998.

[6] K. A. Feiste, B. J. Ronchetti, and D. J. Shippy. System for store forwarding assigning load and store instructions to groups and reorder queues to keep track of program order, 2002. U.S. Patent Number 6,349,382.

[7] M. Franklin. Multi-version caches for Multiscalar processors. In *Proceedings of the First International Conference on High Performance Computing (HiPC)*, 1995.

[8] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *19th International Symposium on Computer Architecture (ISCA-19)*, pages 58–67, Gold Coast, Australia, May 1992.

[9] M. Franklin and G. S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Trans. Comput.*, 45(5):552–571, May 1996.

[10] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 183–193, San Jose, California, Oct. 1994.

[11] A. Gandhi, H. Akkary, R. Rajwar, S. T. Srinivasan, and K. Lai. Scalable load and store processing in latency tolerant processors. In *ISCA 32*, pages 446–457, 2005.

[12] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing*, 1991.

[13] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *MICRO-31*, pages 216–225, 1998.

[14] T. Knight. An architecture for mostly functional languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 88–93, Aug. 1986.

[15] D. Leibholz and R. Razdan. The Alpha 21264: A 500 MHz out-of-order execution microprocessor. In *IEEE COMPCON 42*, 1997.

[16] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *ISCA-24*, pages 181–193, 1997.

[17] A. Moshovos and G. S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *MICRO-30*, 1997.

[18] S. Onder and R. Gupta. Dynamic memory disambiguation in the presence of out-of-order store issuing. In *MICRO-32*, pages 170–176, 1999.

[19] I. Park, C. L. Ooi, and T. N. Vijaykumar. Reducing design complexity of the load/store queue. In *MICRO 36*, pages 411–422, 2003.

[20] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.*, 1(1):3–23, Feb. 1983.

[21] A. Roth. Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization. In *ISCA 32*, pages 458–468, 2005.

[22] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable hardware memory disambiguation for high ilp processors. In *MICRO 36*, pages 399–410, 2003.

[23] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *22nd International Symposium on Computer Architecture*, pages 414–425, Santa Margherita Ligure, Italy, June 1995.

[24] E. F. Torres, P. Ibanez, V. Vinals, and J. M. Llaberia. Store buffer design in first-level multibanked data caches. In *ISCA 32*, pages 469–480, 2005.

[25] G. S. Tyson and T. M. Austin. Improving the accuracy and performance of memory communication through renaming. In *MICRO-30*, 1997.

[26] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, 1996.